School of Computing and Information Systems

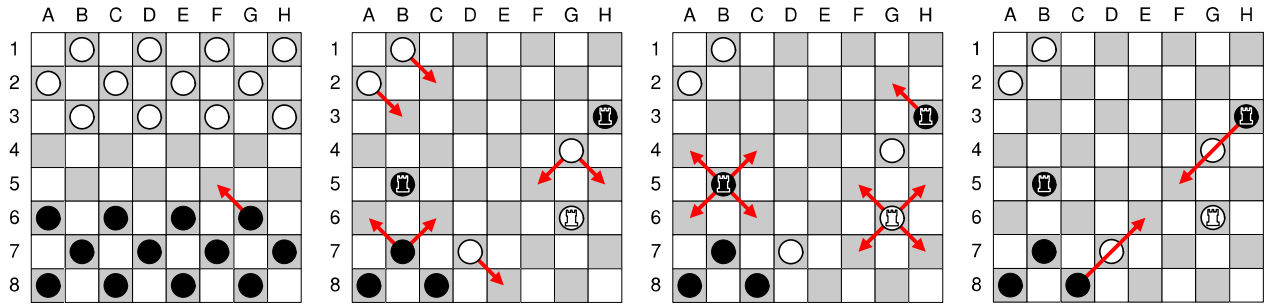
comp10002 Foundations of Algorithms
Semester 2, 2021
Assignment 2

Learning Outcomes

In this project, you will demonstrate your understanding of dynamic memory and linked data structures (Chapter 10) and extend your program design, testing, and debugging skills. You will also learn about Artificial Intelligence and tree search algorithms, and implement a simple algorithm for playing checkers.

Checkers

Checkers, or draughts, is a strategy board game played by two players. There are many variants of checkers. For a guide to checker's families and rules, see `https://www.fmjd.org/downloads/Checkers_families_and_rules.pdf`. Your task is to implement a program that reads, prints, and plays *our* variant of the game.



(a) Initial setup (b) Legal piece moves (c) Legal tower moves (d) Legal captures

Figure 1: Example board configurations, moves, and captures.

Setup. An 8x8 chessboard with 12 black and 12 white *pieces* initially positioned as shown in Figure 1a.

Gameplay. Each player plays all pieces of the same color. Black open the game by making a move, then white make a move, and then players alternate their turns. In a single *turn*, the player either makes a *move* or *capture*. For example, the arrow in Figure 1a denotes an opening move of the black piece from cell `G6` to cell `F5`.

Moves. A piece may move to an empty cell diagonally forward (toward the opponent; north for black and south for white) one square. The arrows in Figure 1b show all the legal moves of black and white pieces.

Towers. When a piece reaches the furthest row (the top row for black and the bottom row for white), it becomes a *tower* (a pile of pieces). The only move of the white piece at cell `D7` in Figure 1b promotes it to the tower. A tower may move to an empty cell diagonally, both forward and backward, one square. The arrows in Figure 1c show all the legal moves of black and white towers.

Captures. To capture a piece or a tower of the opponent, a piece or a tower of the player *jumps* over it and lands in a straight diagonal line on the other side. This landing cell must be empty. When a piece or tower is captured, it is removed from the board. Only *one* piece or tower may be captured in a single jump, and, in our variant of the game, only *one* jump is allowed in a single turn. Hence, if another capture is available after the first jump, it cannot be taken in this turn. Also, in our variant of the game, if a player can make a move or a capture, they may decide which of the two to complete. A piece always jumps forward (toward the opponent), while a tower can jump forward and backward. The arrows in Figure 1d show all the legal captures for both players.

Game end. A player wins the game if it is the opponent's turn and they cannot take *action*, move or capture, either because no their pieces and towers are left on the board or because no legal move or capture is possible.

Input Data

Your program should read input from `stdin` and write output to `stdout`. The input should list actions, one per line, starting from the initial setup and a black move. The input of moves and captures *can* be followed by a single command character, either '`A`' or '`P`'. Action should be specified as a pair of the source cell and the target cell, separated by the minus character '`-`'. For example, "`G6-F5`" specifies the move from cell `G6` to cell `F5`.

The following file `test1.txt` uses eleven lines to specify ten actions, followed by the '`A`' command.

```
1 G6-F5
2 H3-G4
3 F5-H3
4 F3-G4
5 E6-F5
6 G4-E6
7 D7-F5
8 G2-F3
9 F7-G6
10 F1-G2
11 A
12
```

Stage 0 – Reading, Analyzing, and Printing Input Data (8/20 marks)

The first version of your program should read input and print the initial setup and all legal actions. The first 42 lines that your program should generate for the `test1.txt` input file are listed below in the two left columns.

```
1  BOARD SIZE: 8x8
2  #BLACK PIECES: 12
3  #WHITE PIECES: 12
4        A   B   C   D   E   F   G   H
5     +---+---+---+---+---+---+---+---+
6   1 | . | w | . | w | . | w | . | w |
7     +---+---+---+---+---+---+---+---+
8   2 | w | . | w | . | w | . | w | . |
9     +---+---+---+---+---+---+---+---+
10  3 | . | w | . | w | . | w | . | w |
11    +---+---+---+---+---+---+---+---+
12  4 | . | . | . | . | . | . | . | . |
13    +---+---+---+---+---+---+---+---+
14  5 | . | . | . | . | . | . | . | . |
15    +---+---+---+---+---+---+---+---+
16  6 | b | . | b | . | b | . | b | . |
17    +---+---+---+---+---+---+---+---+
18  7 | . | b | . | b | . | b | . | b |
19    +---+---+---+---+---+---+---+---+
20  8 | b | . | b | . | b | . | b | . |
21    +---+---+---+---+---+---+---+---+
```

```
22 ====================================
23 BLACK ACTION #1: G6-F5
24 BOARD COST: 0
25       A   B   C   D   E   F   G   H
26    +---+---+---+---+---+---+---+---+
27  1 | . | w | . | w | . | w | . | w |
28    +---+---+---+---+---+---+---+---+
29  2 | w | . | w | . | w | . | w | . |
30    +---+---+---+---+---+---+---+---+
31  3 | . | w | . | w | . | w | . | w |
32    +---+---+---+---+---+---+---+---+
33  4 | . | . | . | . | . | . | . | . |
34    +---+---+---+---+---+---+---+---+
35  5 | . | . | . | . | . | . | b | . |
36    +---+---+---+---+---+---+---+---+
37  6 | b | . | b | . | b | . | . | . |
38    +---+---+---+---+---+---+---+---+
39  7 | . | b | . | b | . | b | . | b |
40    +---+---+---+---+---+---+---+---+
41  8 | b | . | b | . | b | . | b | . |
42    +---+---+---+---+---+---+---+---+
```

```
1 ====================================
2 *** BLACK ACTION #87: A6-B5
3 BOARD COST: 2
4        A   B   C   D   E   F   G   H
5     +---+---+---+---+---+---+---+---+
6   1 | . | . | . | . | . | . | . | . |
7     +---+---+---+---+---+---+---+---+
8   2 | . | . | . | . | . | . | . | . |
9     +---+---+---+---+---+---+---+---+
10  3 | . | . | . | . | . | . | . | . |
11    +---+---+---+---+---+---+---+---+
12  4 | . | . | . | . | . | . | . | . |
13    +---+---+---+---+---+---+---+---+
14  5 | . | B | . | . | . | . | . | . |
15    +---+---+---+---+---+---+---+---+
16  6 | . | . | w | . | . | . | . | . |
17    +---+---+---+---+---+---+---+---+
18  7 | . | w | . | . | . | . | . | . |
19    +---+---+---+---+---+---+---+---+
20  8 | . | . | b | . | . | . | . | . |
21    +---+---+---+---+---+---+---+---+
```

Lines 1–21 report the board configuration and specify the initial setup from Figure 1a. We use '`b`' and '`w`' characters to denote black and white pieces, respectively. Then, lines 22–42 print the first move specified in the first line of the input. The output of each action starts with the delimiter line of 37 '`=`' characters; see line 22. The next two lines print information about the action taken and the *cost* of the board; see lines 23 and 24. The cost of a board is computed as $b + 3B - w - 3W$, where b, w, B, and W are, respectively, the number of black pieces, white pieces, black towers, and white towers on the board; that is, a tower costs three pieces. Then, your program should print the board configuration that results from the action. The complete output your program should generate in Stage 0 for the `test1.txt` input file is provided in the `test1-out.txt` output file.

If an illegal action is encountered in the input, your program should select and print one of the following six error messages. Your program should terminate immediately after printing the error message.

```
1 ERROR: Source cell is outside of the board.
2 ERROR: Target cell is outside of the board.
3 ERROR: Source cell is empty.
4 ERROR: Target cell is not empty.
5 ERROR: Source cell holds opponent's piece/tower.
6 ERROR: Illegal action.
```

The conditions for the errors are self-explanatory and should be evaluated in the order the messages are listed. Only the first encountered error should be reported. For example, if line 2 of the `test1.txt` file is updated to state "`G2-A8`", line 43 of the corresponding output should report: "`ERROR: Target cell is not empty.`"

Stage 1 – Compute and Print Next Action (16/20 marks)

If the '`A`' command follows the input actions (see line 11 in the `test1.txt` file), your program should compute and print the information about the next Action of the player with the turn. All of the Stage 0 output should be retained. To compute the next action, your program should implement the *minimax decision rule* for the tree depth of *three*. Figure 2 exemplifies the rule for the board configuration in Figure 3a and the turn of black.

First, the tree of all reachable board configurations starting from the current configuration (level 0) and of the requested depth is constructed; if the same board is reachable multiple times in the same tree, the corresponding tree node must be replicated. For example, black can make two moves in Figure 3a: the tower at `A6` can move to `B5` (Figure 3b) and the piece at `C8` can move to `D7` (Figure 3c); see level 1 in Figure 2. The tree in Figure 2 explicitly shows nodes that refer to 15 out of all 30 board configurations in the minimax tree of depth three for the black turn and the board from Figure 3a. The labels of the nodes refer to the corresponding boards shown in Figure 3. For instance, nodes with labels (f)–(h) at level 2 of the tree refer to the boards in Figures 3f to 3h, respectively, which are all the boards white can reach by making moves and captures in the board in Figure 3c.

Second, the cost of all leaf boards is computed; see the nodes highlighted with a gray background (level 3). For example, six board configurations at level 3 of the tree can be reached from the board in Figure 3d. The boards in Figures 3i and 3j have the cost of 3, while the four boards reachable via all moves of the tower at cell `B5` in board (d), not shown in the figure, have the cost of 2. Intuitively, a positive cost suggests that black win, a negative cost tells that white win, and the magnitude of the cost indicates the advantage of one player over the other.

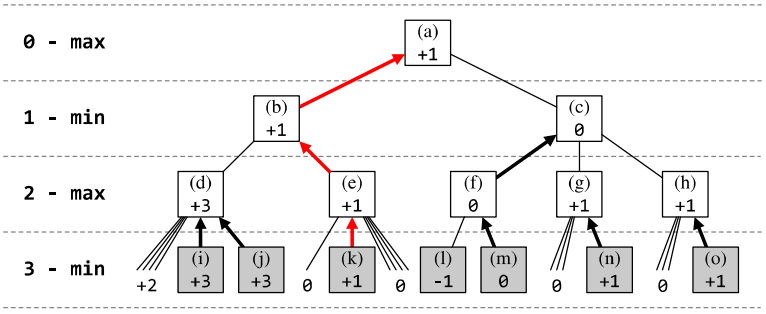


Figure 2: A minimax tree.

Third, for each possible action of the player, we check all possible actions of the opponent, and choose the next action of the player to be the first action on the path from the root of the tree toward a leaf node for which the player maximizes their gain while the opponent (considered rational) aims to minimize the player's gain; see the red path in Figure 2. Black take actions in the boards at level 2 of the tree. At this level, black aim to maximize the gain by choosing an action toward a board of the highest cost. This is cost 3 for board (d) toward boards (i) and (j), 1 for (e) toward (k), 0 for (f) toward (m), 1 for (g) toward (n), and 1 for (h) toward (o); the arrows from level 3 to level 2 encode the cost selections and node labels at level 2 encode propagated costs. White take actions in the boards at level 1 of the tree. White aim to maximize their gain, which translates into minimization of the gain by black. Thus, at every board at level 1, white choose an action toward a board at level 2 with the lowest cost propagated from level 3. This is cost 1 for board (b) toward (e), and 0 for (c) toward (f); again, see the arrows from level 2 to level 1 and the costs as node labels at level 1. Finally, to maximize their gain, black pick the next action in the game to be the one that leads to the board with the highest propagated cost at level 1. This is move "`A6-B5`" toward board (b) on the path to (k); assuming white are rational and play "`B7-A8`" next.

To compute the next action for white, the order of max- and min-levels must be flipped. If several children of the root have the same propagated maximum/minimum cost, the action to the left-most such child must be chosen. To construct the children, both for black and white turns, the board is traversed in *row-major order*

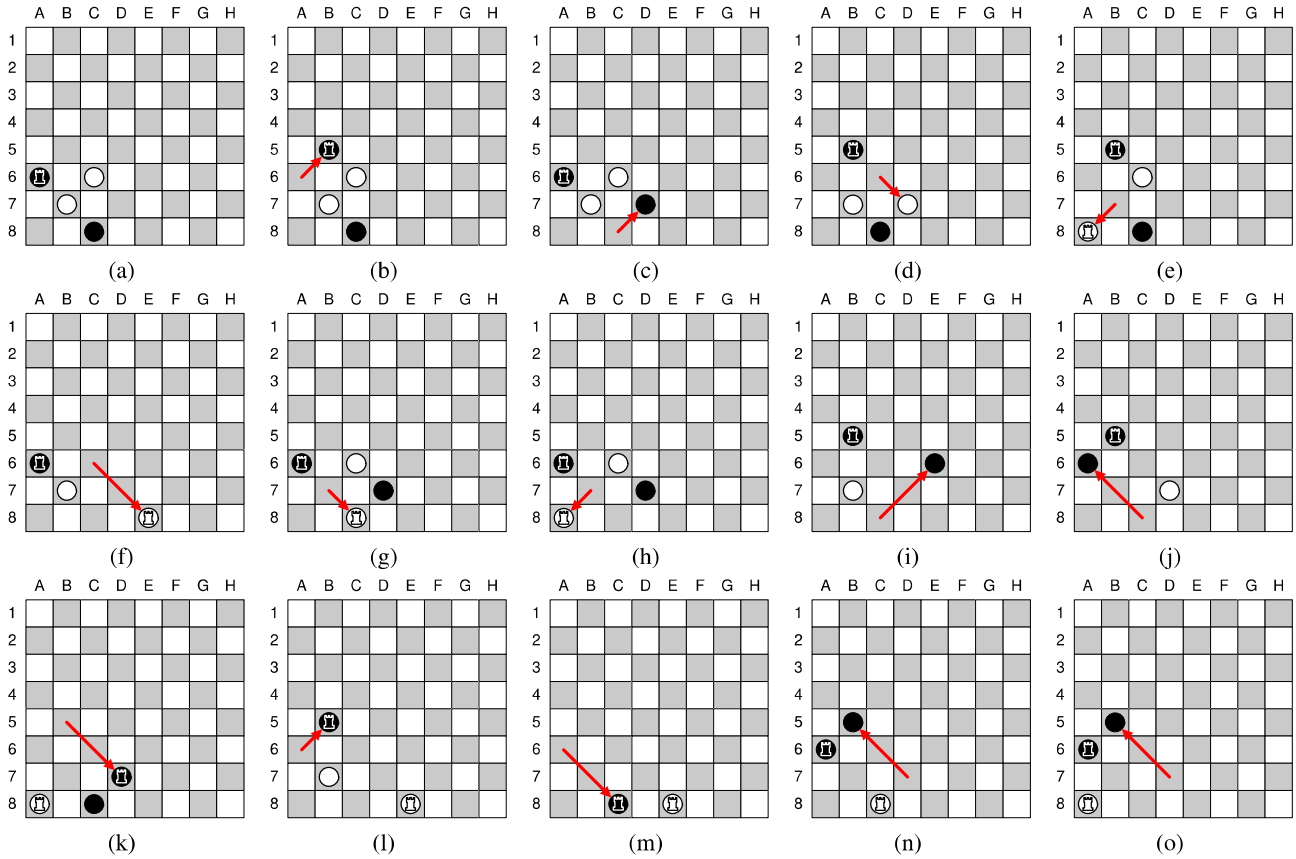


Figure 3: Board configurations for possible evolutions of an example checkers endspiel.

and for each encountered piece or tower all possible actions are explored, starting from the north-east direction and proceeding clockwise. Children should be added from left to right in the order they are constructed. Each computed action should be printed with the "`***`" marker. Lines 1–21 in the right column of the listing in the Stage 0 description show the output for the example computed action. Black and white towers are denoted by '`B`' and '`W`' characters, respectively. Boards in which black or white cannot take an action cost `INT_MIN` and `INT_MAX`, respectively, defined in the `<limits.h>` library. If the next action does not exist and, thus, cannot be computed, the player loses, and the corresponding message is printed on a new line ("`BLACK WIN!`" or "`WHITE WIN!`").

Stage 2 – Machines Game (20/20 marks)

If the '`P`' command follows the input actions, your program should `Play` *ten* next actions or all actions until the end of the game, whatever comes first. The game should continue from the board that results after processing the Stage 0 input. If the game ends within the next ten turns (including the last turn when no action is possible), the winner should be reported. The computation of actions and winner reporting should follow the Stage 1 rules.

Important...

The outputs generated by your program should be exactly the same as the sample outputs for the corresponding inputs. Use `malloc` and dynamic data structures of your choice to implement the minimax rule for computing the next action. Before your program terminates, all the `malloc`'ed memory must be `free`'d.

Boring But Important...

This project is worth 20% of your final mark, and is due at **6:00pm on Friday 15 October**.

Submissions that are made after the deadline will incur penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to obtain a letter from them that describes your illness and their recommendation for treatment. Suitable documentation should be attached to **all** extension requests.

You need to submit your program for assessment **via the LMS Assignment 2** page. Submission is **not possible through grok**. There is a pre-submission test link also provided on the LMS page, so you can try your program out in the test environment. **It does not submit your program for marking either.** Note that this pre-test service will likely overload and fail on the assignment due date, and if that does happen, it will not be a basis for extension requests. *Plan to start early and to finish early!!*

Multiple submissions may be made; only the last submission that you make before the deadline will be marked. If you make any late submission at all, your on-time submissions will be ignored, and if you have not been granted an extension, the late penalty will be applied. A rubric explaining the marking expectations is linked from the LMS, and you should study it carefully. Marks will be available on the LMS approximately two weeks after submissions close, and feedback will be mailed to your student email account.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** have any "accidents" that allow others to access your work; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrolment terminated for such behavior.

The FAQ page contains a link to a program skeleton that includes an Authorship Declaration that you must "sign" and include at the top of your submitted program. Marks will be deducted (see the rubric linked from the FAQ page) if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action.

Nor should you post your code to any public location (`github`, `codeshare.io`, etc) while the assignment is active or prior to the release of the assignment marks.